DENTRO LA SCATOLA

Rubrica a cura di

Fabio A. Schreiber

Il Consiglio Scientifico della rivista ha pensato di attuare un'iniziativa culturalmente utile presentando in ogni numero di Mondo Digitale un argomento fondante per l'Informatica e le sue applicazioni; in tal modo, anche il lettore curioso, ma frettoloso, potrà rendersi conto di che cosa sta "**dentro la scatola**". È infatti diffusa la sensazione che lo sviluppo formidabile assunto dal settore e di conseguenza il grande numero di persone di diverse estrazioni culturali che - a vario titolo - si occupano dei calcolatori elettronici e del loro mondo, abbiano nascosto dietro una cortina di nebbia i concetti basilari che lo hanno reso possibile.
Il tema scelto per il 2004 è stato: "***Perché gli anglofoni lo chiamano computer***, ovvero: ***introduzione alle aritmetiche digitali***". Per il 2005 il filo conduttore della serie sarà: "***Ma ce la farà veramente?***, ovvero: ***introduzione alla complessità computazionale e alla indecidibilità***" e il suo intento è di guidare il lettore attraverso gli argomenti fondanti dell'Informatica e alle loro implicazioni pratiche e filosofiche. La realizzazione degli articoli è affidata ad autori che uniscono una grande autorevolezza scientifica e professionale a una notevole capacità divulgativa.

Formule, numeri e paradossi

Dino Mandrioli

1. Introduzione

Questo è il secondo articolo della serie dedicata agli aspetti più concettuali e teorici dell'informatica. Esso è in diretta relazione con il primo [1] e, per una lettura e comprensione non superficiali, ne presuppone la conoscenza. In [1] infatti abbiamo visto come il concetto fondamentale di tutta l'informatica, l'algoritmo, inteso come processo di calcolo automatizzabile, abbia radici in comune con la matematica e la filosofia, che risalgono a diversi secoli prima di Cristo. Abbiamo inoltre osservato come la classe dei problemi risolvibili automaticamente, ossia mediante algoritmi, pur vastissima, presenti importanti e "incolmabili lacune": molti problemi di notevole rilevanza pratica e concettuale possono essere risolti –*se possono essere risolti*- solo ricorrendo a doti umane difficilmente formalizzabili come intuizione, esperienza, ...
Questo secondo articolo, di taglio necessariamente più tecnico-matematico[1] fornirà una dimostrazione dell'enunciato suddetto; il suo scopo, tuttavia, più che aumentare nel lettore il convincimento dell'enunciato stesso, è mostrare come le tecniche utilizzate dai "pionieri" per giungere a questi risultati abbiano molto in comune con altre forme classiche di ragionamento matematico e filosofico e mostrare quindi –nuovamente- i profondi legami che storicamente uniscono queste discipline.

2. L'enumerazione (o Goedelizzazione)

Il termine "enumerazione" è di largo uso nel linguaggio comune: si possono enumerare le squadre di un campionato, gli alunni di una classe, le offerte commerciali di una ditta ecc..
In termini matematici enumerare gli elementi di un insieme *S* significa stabilire una *corrispondenza biunivoca* tra *S* e l'insieme dei numeri naturali, o un suo sottoinsieme, se il numero degli elementi di *S* è finito. Si noti infatti che la seguente enumerazione dei giorni della settimana, {lunedì, martedì, ... domenica} implicitamente associa all'elemento "lunedì" il numero 1, a "martedì" il numero 2 e così via. In

[1] Lo stile espositivo, tuttavia, sarà il più possibile informale e discorsivo. Il lettore interessato ad entrare approfonditamente negli aspetti più formali può consultare, per esempio [2].

questo modo possiamo riferirci al "terzo giorno della settimana" per intendere "mercoledì". In generale, una qualsiasi tabella costituisce un'enumerazione.
Un insieme si dice perciò numerabile, o enumerabile, se è possibile stabilire una corrispondenza biunivoca tra esso e un sottoinsieme dei numeri naturali. Non a caso linguaggi di programmazione come il Pascal permettono di usare come indice di un array (che altro non è che una tabella finita) un qualsiasi insieme numerabile finito: è infatti immediato "tradurre" un indice "giorno della settimana" in un numero compreso tra 1 e 7 e viceversa (d'ora in avanti però, per uniformarci alla prassi del linguaggio *C* e dei suoi derivati nonché al fatto che i numeri naturali comprendono anche lo 0, inizieremo sempre le nostre enumerazioni dallo 0).
Moltissimi, anzi infiniti insiemi sono numerabili[2]. Per esempio, fissato un alfabeto di caratteri, l'insieme delle stringhe di lunghezza finita di tali caratteri è numerabile in varie maniere: supponendo per semplicità che l'alfabeto in questione contenga i soli caratteri 'a' e 'b', una possibile enumerazione è per lunghezza crescente e in ordine alfabetico: {ε, a, b, aa, ab, ba, bb, aaa, aab, ...}, dove ε denota la stringa convenzionale di lunghezza 0, ossia costituita da 0 caratteri, detta anche stringa vuota o stringa nulla.
Essendo poi un linguaggio (sia naturale, come l'Italiano, che artificiale, come un linguaggio di programmazione) un sottoinsieme dell'insieme di tutte le stringhe ottenibili da un dato alfabeto, se ne deduce che ogni linguaggio è a sua volta numerabile.
Un caso particolare di linguaggio e di relativa enumerazione è costituito dall'insieme delle Macchine di Turing. Facendo riferimento alla descrizione di questo modello fornita in [1], un semplice modo per enumerare tutte le macchine di Turing è il seguente. Scriviamo prima tutte le possibili macchine con due stati (macchine a uno stato solo non sono molto significative, ma, volendo, nulla ci impedisce di comprenderle nel nostro elenco; rammentiamo anche che, senza perdere in generalità abbiamo adottato un alfabeto costituito da due soli simboli: lo fanno anche tutti i calcolatori di questo mondo ...). Orbene le possibili macchine con due stati sono in un numero finito: c'è un numero finito di modi diversi di riempire le 4 caselle corrispondenti alle 2×2 combinazioni possibili di stati e simboli dell'alfabeto mediante terne del tipo <simbolo da scrivere sul nastro, nuovo stato, spostamento della testina>[3]. Possiamo perciò ordinarle, ossia enumerarle, secondo un qualche criterio; per esempio, possiamo indicare come "macchina numero 0" quella con tutte le caselle vuote, come "macchina numero 1" quella in cui l'unica casella non vuota è quella in basso a sinistra e che contiene la terna <_, s1, *R*>[4], come "macchina numero 2" quella in cui l'unica casella non vuota è quella in basso a sinistra e che contiene la terna <_, s1, *L*> ecc.. Dopo aver enumerato l'ultima macchina con due stati (per esempio quella che ha tutte le caselle piene e contenenti la terna <|, s2, *N*>, passiamo ad enumerare tutte le macchine con 3 stati (saranno molte più di quelle a due stati ma sempre in numero finito) e così via: ad ogni macchina abbiamo assegnato un numero d'ordine appartenente all'insieme dei numeri naturali.
Questo procedimento ci suggerisce due osservazioni fondamentali:
1. Il procedimento è assolutamente generale e può essere facilmente esteso a qualsiasi "linguaggio" nell'accezione più ampia possibile del termine: allo stesso modo sono perciò enumerabili tutti i programmi *C* o Pascal, tutte le formule che si possono scrivere in logica matematica, tutti gli spartiti musicali ecc..
2. Il *procedimento* è anche *algoritmico*. Infatti non dovrebbe essere difficile riconoscere nella sua descrizione soprastante i requisiti tipici di un algoritmo. Chiunque abbia un po' di esperienza di programmazione potrebbe ricavarne un programma che, ricevuta in ingresso la descrizione di una macchina di Turing, attraverso la sua tabella, produca in uscita il numero ad essa corrispondente e *viceversa*.
Un tale processo di enumerazione algoritmica

[2] Non tutti però! Anzi vedremo tra breve che ancor di più non lo sono.
[3] Per essere precisi questo numero è $(2.2.3 + 1)^{(2.2)} = 13^4$. Perché?
[4] I simboli *R*, *L*, *N* indicano, rispettivamente, spostamento a destra, a sinistra e non spostamento della testina.

degli elementi di un insieme si dice anche *Goedelizzazione*, in omaggio a Kurt Goedel che lo usò sistematicamente per ricavare i suoi fondamentali risultati già menzionati in [1]. Perciò l'indice associato dall'enumerazione a un generico elemento *X* dell'insieme viene anche chiamato numero di *Goedel* di *X*. Per esempio, se nella precedente enumerazione una macchina occupa la posizione *i*, la indicheremo con M_i e diremo che essa è la *i*-esima macchina di Turing e che *i* è il suo numero di Goedel.

L'enumerazione è anche una tecnica, forse banale, ma semplice e spesso efficace, per risolvere molti problemi. Per esempio, se vogliamo stabilire se un oggetto si trova in un insieme (tabella, array, o altra struttura dati) la cosa più semplice da fare è enumerarne tutti gli elementi e confrontarli, uno per uno, con l'oggetto in questione: se il confronto dà esito positivo in qualche caso ne otteniamo una risposta positiva; altrimenti, se giungiamo al termine dell'enumerazione senza aver trovato alcun elemento identico a quello in questione, la ricerca dà esito negativo. Similmente, se vogliamo calcolare la radice quadrata intera di un numero naturale *n*, possiamo enumerare tutti i numeri naturali *m*, iniziando da 0 o da 1; calcolarne il quadrato e confrontarlo con *n*; non appena troviamo un valore *m*, tale che $m^2 > n$, possiamo concludere che $m - 1$ è la radice cercata.

Consideriamo ora il *decimo problema di Hilbert*, ossia il problema di stabilire se un polinomio a coefficienti interi in un qualsiasi numero di variabili ammetta radici intere. In [1] abbiamo affermato che questo problema è stato dimostrato indecidibile. Osserviamo tuttavia che, data una qualsiasi *n*-pla di valori interi, è immediato verificare se essa è una radice di un certo polinomio in *n* variabili. Potremmo perciò enumerare tutte le *n*-ple (per esempio, per $n = 3$, una possibile enumerazione è {<0,0,0>; <0,0,1>; <0,0,-1>; <0,1,0>; <0,-1,0>; ...<0,1,1>; ....}) e per ognuna di esse verificare se essa è una radice del polinomio. In questa maniera, evidentemente, *se una radice intera esiste*, prima o poi (magari dopo tempi biblici, ma questo aspetto al momento non ci riguarda) la si trova. Il punto critico però sta nella domanda: "e se una tale radice non esiste?". Evidentemente in tal caso continueremmo ad enumerare *n*-ple di valori interi (che sono infinite) senza poterci mai arrestare. Siamo tornati al problema della terminazione del calcolo: come fare per sapere se la nostra enumerazione prima o poi avrà successo? Il fatto che il decimo problema di Hilbert sia indecidibile significa che non c'è modo di saperlo (algoritmicamente). Tuttavia, il procedimento di cui sopra ha tutti i requisiti dell'algoritmo; diversamente dagli altri algoritmi considerati finora, però, la sua esecuzione termina solo se il problema ammette soluzione.

Per questo motivo problemi come questo, per i quali esistono algoritmi che calcolano la soluzione, se essa esiste, ma in caso contrario non forniscono garanzia di terminazione, si dicono *semidecidibili*. Si noti che lo stesso problema della terminazione del calcolo è banalmente semidecidibile: infatti basta "far girare la macchina *M* sul dato *x*" per scoprire che la sua esecuzione terminerà ... *se terminerà*!

Purtroppo però, tra semidecidibilità e decidibilità c'è di mezzo ... l'altra metà come ci dimostreranno le sezioni 4 e 5.

3. Il problema dell'halt connessioni con matematica e filosofia

Siamo ora in grado di provare quanto affermato in [1], ossia "*Non esistono algoritmi, ossia macchine di Turing, in grado di risolvere il problema della terminazione del calcolo*".

La dimostrazione rigorosa è data nel riquadro. A prima vista la dimostrazione potrebbe sembrare una "tipica, noiosa dimostrazione di un tipico teorema che può interessare pochi strampalati amanti della matematica". In realtà essa ripercorre alcuni fondamentali paradigmi del ragionamento umano che, come abbiamo già affermato nell'articolo precedente [1], sono serviti, nel corso dei millenni, ad evidenziare proprio i limiti del nostro ragionamento. Per fissare le idee, potremmo enunciare questi paradigmi con i termini seguenti: *assurdo, diagonalizzazione, negazione*. Il ragionamento per assurdo è il più evidente e non dovrebbe richiedere ulteriori spiegazioni e commenti. Concentriamoci perciò sugli altri due.

Il procedimento utilizzato è *diagonale*, perché abbiamo considerato un dominio di coppie di numeri <*m*, *x*>, rappresentabili in un piano cartesiano a coordinate intere e ci siamo posti sul-

Teorema dell'Halt

In primo luogo riformuliamo il problema in termini "numerici". Grazie alle tecniche enumerative, un qualsiasi dominio di dati può essere rappresentato attraverso l'insieme dei numeri naturali, $\mathcal{N}$. Quindi un qualsiasi problema può essere formalizzato come una funzione f, con dominio $\mathcal{N}$ e codominio $\mathcal{N}$. Una macchina di Turing M che quindi calcola una funzione f, può essere a sua volta indicata attraverso il suo numero di Goedel m. Denotiamo con f_m la funzione calcolata da M_m. Poiché sappiamo che in generale una macchina M potrebbe non terminare mai la sua computazione in corrispondenza di un dato di ingresso x, diciamo allora che in tal caso $f_m(x)$ è *indefinita*, e indichiamo questo fatto con la notazione $f_m(x) = \perp$, dove il simbolo $\perp$ indica appunto un valore convenzionale indefinito, ovviamente non appartenente ad $\mathcal{N}$.
Il problema di decidere se la macchina M terminerà la sua esecuzione in corrispondenza del dato di ingresso x significa perciò stabilire se, per un generico m e un generico x, $f_m(x) \neq \perp$ o no. Orbene, ragioniamo per assurdo e supponiamo che esso sia decidibile. Ciò equivale a dire che esiste un algoritmo, ossia una macchina di Turing, che calcola la seguente funzione:

$g(m, x) = 1$ se $f_m(x) \neq \perp$; $g(m, x) = 0$ se $f_m(x) = \perp$

Ma se una tale macchina esistesse sarebbe molto facile ricavare da essa un'altra macchina che calcola invece la funzione:

$h(z) = 1$ se $f_z(z) = \perp$; $h(z) = \perp$ se $f_z(z) \neq \perp$

Per calcolare $h(z)$ per un generico z basterebbe infatti calcolare la funzione g per $m = x = z$; se il risultato fosse $g(z, z) = 0$, produrre in uscita il valore 1; se invece fosse 1 basterebbe portare la macchina in uno stato in cui, per esempio, la testina continua a spostarsi a destra di una posizione senza cambiare più stato, e fare in modo che la macchina non si arresti mai. Dunque, in base alla nostra ipotesi (assurda), esiste una macchina che calcola h; essa avrà un numero di Goedel, diciamo y. Quindi, in base alla nostra notazione, h è la funzione f_y. Domandiamoci allora "Quanto vale $h(y)$?"
Per come è definita, $h(y)$ può solo valere 1 oppure essere ... indefinita (il lettore ci scuserà il gioco di parole, pur matematicamente corretto).
Supponiamo dunque che sia $h(y) = 1$. Essendo h la funzione f_y, ciò significa $f_y(y) = 1$; ma, se nella precedente definizione poniamo $z = y$, otteniamo anche $f_y(y) = \perp$: una contraddizione. Dobbiamo perciò escludere la possibilità $h(y) = 1$. Non resta che esaminare l'unica alternativa possibile, ossia $h(y) = \perp$. Ripetendo il ragionamento però, ciò significa $f_y(y) = \perp$ ma anche il suo contrario, $f_y(y) \neq \perp$. Nuovamente, una contraddizione, che quindi conclude il ragionamento per assurdo. QED.

la diagonale del piano, ossia abbiamo considerato il caso $m = x = z$.
Il procedimento utilizzato è *centrato sulla negazione*, perché "abbiamo scambiato il Sì con il No": ciò che era 1 nella definizione di g è diventato $\perp$ nella definizione di h mentre lo 0 di g è diventato 1 per h.
Orbene questi stessi "schemi mentali" si ritrovano in numerosi *paradossi* elaborati da diversi filosofi o matematici nel corso dei secoli. Per esempio, il "paradosso del barbiere" definisce l'unico barbiere di un paese come colui che rade tutti e solo coloro che non si radono da soli, per ricavarne che il barbiere non può né radersi da solo né non radersi da solo: se indichiamo con $g(m, x) = 1$ il fatto che m rade x e con $g(m, x) = 0$ il fatto che m non rade x, il "non radersi da solo", palesemente corrisponde al valore $g(z, z) = 0$. Simili analogie valgono per il paradosso di Russel, che fa riferimento all'"insieme degli insiemi che non appartengono a se stessi", per concludere che un tale insieme non può né appartenere a se stesso né non appartenere a se stesso e mettere così in crisi l'insiemistica tradizionale.
Ancora più evidente è l'analogia con la dimostrazione del teorema di Cantor che afferma che non è possibile stabilire una corrispondenza biunivoca tra $\mathcal{N}$ e il suo insieme delle parti $\mathcal{P}(\mathcal{N})$[5]. Anche in questo caso infatti, si assume per assurdo che una tale corrispondenza esista, ossia che a ogni numero n corrisponda uno e un solo sottoinsieme di $\mathcal{N}$, S_n, e, viceversa, ogni sottoinsieme S di $\mathcal{N}$ "abbia un suo indice" ossia compaia nella enumerazione $\{S_n\}$. Definiamo allora quell'insieme $S \subseteq \mathcal{N}$, costituito da tutti quei numeri i tali che $i \notin S_i$. S è ben definito, se vale l'ipotesi assurda: infatti per esempio, $0 \in S$ se e solo se $0 \notin S_0$, $1 \in S$ se e solo se $1 \notin S_1$ ecc[6].. Allora un tale S (che potrebbe anche essere l'insieme vuoto o

[5] Sul teorema di Cantor è fondato il concetto di cardinalità degli insiemi infiniti. Grazie ad esso, si può affermare che i numeri reali hanno la "cardinalità del continuo", che è superiore alla "cardinalità del numerabile" che accomuna invece numeri naturali, numeri interi, numeri razionali, nonostante gli uni siano sottoinsiemi propri degli altri.

[6] Si noti che qui la computabilità non c'entra: stiamo *definendo* S indipendentemente dal fatto che esista un *algoritmo per decidere* se un generico $n \in S$.

l'insieme universo) deve trovarsi nell'enumerazione $\{S_n\}$; ossia deve esistere un m tale che $S = S_m$. A questo punto il gioco è chiaro: $m \in S$? Evidentemente sia la risposta *Sì* che la risposta *No* portano ad una contraddizione e quindi alla conclusione dell'assurdità dell'ipotesi di partenza.
Dal teorema di Cantor possiamo anche ricavare un significativo corollario del fatto che esistano problemi non risolvibili algoritmicamente.
I problemi non risolvibili algoritmicamente sono "*molti di più di quelli risolvibili algoritmicamente*". Essi hanno infatti la cardinalità del continuo mentre quelli risolvibili sono un insieme numerabile.
Per rendercene conto ricordiamo che il concetto di problema può essere formalizzato, tra l'altro, come una funzione con dominio e codominio $\mathcal{N}$. L'insieme di tali funzioni contiene l'insieme delle funzioni con dominio $\mathcal{N}$ e codominio {0, 1} (essendo questo un sottoinsieme di $\mathcal{N}$) che sono evidentemente in corrispondenza biunivoca con $\mathcal{P}(\mathcal{N})$ (ad un generico $S \subseteq \mathcal{N}$ faccio corrispondere la funzione $f(x) = 1$ se $x \in S$, $f(x) = 0$ se $x \notin S$); e $\mathcal{P}(\mathcal{N})$, abbiamo visto, non è numerabile (ovviamente è "più che numerabile", ossia ha una cardinalità maggiore della cardinalità dei numeri naturali). Al contrario, per definizione, l'insieme dei problemi risolvibili non può essere "più grande" dell'insieme delle macchine di Turing (o dei programmi *C*) che invece è numerabile.
Si noti tuttavia che questo risultato ha una portata più concettuale che pratica: infatti è bensì vero che "l'insieme dei problemi non risolvibili sta all'insieme dei problemi risolvibili come i numeri irrazionali stanno ai numeri razionali", però la "gran parte" di questi problemi (ossia, ancora una quantità non numerabile) non è neanche *esprimibile*. Infatti per definire un problema abbiamo bisogno di un meccanismo espressivo, ossia di un *linguaggio*. E un linguaggio altro non è che un sottoinsieme di tutte le stringhe ottenibili a partire da un alfabeto base di caratteri (in senso lato). Tale insieme, abbiamo visto, è numerabile. Quindi l'insieme dei *problemi definibili*, è un insieme numerabile, esattamente come l'insieme dei *problemi risolvibili*. Sfortunatamente il secondo è un sottoinsieme proprio del primo, esattamene come l'insieme dei numeri pari è un sottoinsieme proprio dell'insieme dei numeri naturali pur avendo la stessa cardinalità.
Chiudiamo infine questa sezione osservando che anche i teoremi di indecidibilità e di incompletezza di Goedel menzionati in [1] sfruttano tecniche dimostrative simili a quelle presentate qui, confermando perciò la grande generalità e valenza concettuale di queste forme di ragionamento.

4. Il problema complemento

Nella sezione 2 abbiamo introdotto il termine "problema semidecidibile", intendendo con esso un problema per il quale esiste un algoritmo che ne calcola la soluzione ... se questa esiste. Più precisamente, posto che il termine decidibilità è sinonimo di "calcolabilità" laddove il problema sia formulato in termini "booleani" (come specificato in [1]), ossia sotto forma di domanda che ammetta una risposta "SI o NO", un problema è decidibile quando esiste un algoritmo in grado di fornire sempre la risposta corretta; un problema è invece semidecidibile quando esiste un algoritmo che, se la risposta è SI, allora la fornisce sempre correttamente, e ... se la risposta è NO? In tal caso, certamente non può dare una risposta sbagliata, ossia SI; allora che risposta potrebbe dare? Potrebbe, ovviamente, fornire la risposta giusta, ossia NO, ma potrebbe anche *non dare alcuna risposta*, ossia *non terminare mai l'esecuzione*.
Che ci sia un'importante differenza tra decidibilità e semidecidibilità è certificato dalla constatazione precedente che il problema dell'halt è semidecidibile ma non è decidibile. Cerchiamo allora di capire da dove deriva questa differenza. Orbene, la differenza sta nel "problema complemento". Come suggerisce il termine, il problema complemento $P^\wedge$ di un generico problema P si ottiene semplicemente da P invertendo le risposte: se la risposta a P era SI la risposta a $P^\wedge$ sia NO e viceversa.
A prima vista non sembrerebbero esserci profonde differenze tra un problema e il suo complemento: ad esempio, se ho un algoritmo che è in grado di rispondere SI o NO alla domanda se un certo valore si trovi in una certa tabella, con una banale modifica ne posso rica-

vare un algoritmo in grado di rispondere SI o NO alla domanda se un certo valore non si trovi in una certa tabella.
Il punto critico di questo ragionamento risiede nel fatto che le due risposte SI, NO possono essere scambiate tra loro se c'è la garanzia che *vengano fornite entrambe al termine della computazione*; ciò richiede dunque la garanzia che la computazione termini sempre, sia quando la risposta è SI che quando la risposta è NO, e questo è proprio ciò che non accade quando un problema è soltanto semidecidibile, ma non decidibile: in tal caso quando la risposta è NO l'algoritmo di (semi)soluzione del problema non è in grado di fornirla perché per farlo dovrebbe "sapere" che la sua computazione non terminerà. Questa constatazione ci permette di sottolineare un dettaglio tutt'altro che marginale nella tecnica che abbiamo utilizzato per dimostrare l'indecidibilità del problema dell'halt: non ci siamo limitati, sulla base dell'ipotesi assurda, a scambiare il SI ($g(z, z) = 1$) con il NO ($g(z, z) = 0$), ma abbiamo anche trasformato la risposta NO in una non-terminazione: altro è il fatto che per definizione un problema abbia risposta negativa se un algoritmo per la sua soluzione non termina e altro è il fatto che tale risposta negativa sia computabile, anche nell'ipotesi che sia computabile la risposta positiva: quando c'è di mezzo la possibilità che il calcolo non termini le risposte SI e NO non sono semplici bit facilmente invertibili con un operatore NOT!
Osserviamo però che il prefisso "semi" nel termine "semidecidibilità" è quanto mai appropriato, in quanto, "due semidecidibilità fanno una decidibilità". Vediamo di spiegare meglio questa battuta.
Se il problema P è semidecidibile ciò significa che esiste un algoritmo A la cui esecuzione, se la risposta a P è SI, prima o poi termina e produce la risposta corretta. Se dunque anche il complemento di P, $P^\wedge$, è semidecidibile, vuole dire che esiste anche un algoritmo $A^\wedge$ che, se la risposta a $P^\wedge$ è SI (cioè la risposta a P è NO), prima o poi termina e produce la risposta corretta. Allora è sufficiente eseguire A e $A^\wedge$ in parallelo ed avere sufficiente pazienza: in tal caso, prima o poi almeno uno dei due terminerà con la risposta corretta e a quel punto potrò tranquillamente sospendere anche l'esecuzione dell'altro.

5. Risvolti pratici e concettuali della semidecidibilità

Nel precedente articolo [1] abbiamo citato diversi esempi di problemi indecidibili; con particolare riferimento alle proprietà dei programmi, è indecidbile, per esempio, la presenza di alcuni tipici errori come il rischio di divisione per 0, l'accesso ad un array attraverso un indice al di fuori dei limiti imposti ecc.. Nella prassi della programmazione questi errori vengono solitamente classificati come "errori a run-time": diversamente dagli "errori a compile-time" che possono essere individuati dal compilatore –la cui presenza, quindi è un problema decidibile-, essi possono essere segnalati (inserendo nel codice oggetto opportune istruzioni di verifica) solo durante l'esecuzione e solo al momento in cui si manifestano.
Ciò è la naturale conseguenza pratica del fatto che questi problemi siano semidecidibili: con un ragionamento che costituisce un'ulteriore estensione della tecnica diagonale illustrata in precedenza si può infatti dimostrare che, se uno di questi errori si annida in un programma, una opportuna attività di testing del programma è in grado, *prima o poi*, di segnalarlo. Qui però occorre distinguere con attenzione il problema dal suo complemento: semidecidibile è la *presenza* dell'errore; siccome lo stesso problema non è decidibile, ne deduciamo che il suo complemento, ossia, l'*assenza dell'errore*, non solo non è decidibile ma non è neanche semidecidibile, altrimenti sia la presenza che l'assenza dell'errore sarebbero anche decidibili. Questa constatazione costituisce il fondamento teorico del famoso slogan coniato da Dijkstra per sottolineare limiti e criticità dell'attività di testing: "Il testing serve per dimostrare la presenza di errori, non potrà mai dimostrare invece la correttezza di un programma". In altri termini, dimostrare la correttezza di un programma è ancor più difficile che dimostrarne la scorrettezza[7].

[7] Con questa osservazione abbiamo "urtato la punta dell'iceberg", ossia la criticità dell'attività di testing per il software. Per sviscerare il tema e investigare anche le differenze con altre tecniche di verifica sperimentale in diversi settori dell'ingegneria, però, dobbiamo rimandare il lettore ad una letteratura più ampia e specializzata (per esempio, [3]).

In modo del tutto analogo, supponiamo di voler stabilire se una certa formula matematica è un teorema oppure no: se, provando diversi casi, ne troviamo un controesempio (ossia un caso che contraddice la formula), possiamo certo concludere la sua falsità; ma tanti tentativi con esito positivo non costituiscono una dimostrazione.
In conclusione, il fatto di non aver trovato la soluzione di un problema dopo un certo numero di tentativi non ci autorizza a concludere che la soluzione non esiste.

Bibliografia

[1] Mandrioli D.: Potenza e limiti del calcolo automatico: le radici teoriche dell'informatica. *Mondo Digitale*, Vol. 4, N.1, p. 64-69, 2005.

[2] Mandrioli D., Ghezzi C.: *Theoretical Foundations of Computer Science*. John Wiley & Sons, 1987. Disponibile anche nella traduzione italiana, edita da UTET, 1989.

[3] Ghezzi C., Jazayeri M., Mandrioli D.: *Fundamentals of Software Engineering*. II edizione, Prentice-Hall, 2002. Disponibile anche nella traduzione italiana, edita da Pearson Education Italia, 2004.

Dino Mandrioli è professore ordinario di Informatica Teorica presso il Politecnico di Milano. I suoi interessi di ricerca sono principalmente nei settori dell'informatica teorica, dell'ingegneria del software e dei sistemi critici in tempo reale. Ha pubblicato oltre 80 articoli scientifici su riviste ed atti di convegni internazionali. È coautore di vari libri, fra cui *Theoretical Foundations of Computer Science* (J. Wiley & Sons), *Fundamentals of Software Engineering* (Prentice-Hall), *The Art and Craft of Computing* (Addison Wesley). Dino Mandrioli è stato membro del Program Committee di diverse conferenze internazionali, Associate Editor di diverse riviste internazionali, program co-chairman della conferenza Formal Methods 2003.
mandrioli@elet.polimi.it